Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 69 (2007) 46–55
Science of Computer Programming
www.elsevier.com/locate/scico

The CommUnity Workbench☆

Cristóvão Oliveira[a,*], Michel Wermelinger[b]

[a] *Dep. de Informática, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal*
[b] *Computing Department, The Open University, Milton Keynes MK7 6AA, UK*

Received 2 November 2005; accepted 16 September 2006
Available online 10 October 2007

Abstract

COMMUNITY is a formal approach to Software Architecture with a strict separation of the computation, coordination, and distribution aspects. The approach is based on a parallel design language with state, which facilitates the specification of computations compared to the process calculi used by other formal approaches, and on category theory, which provides an intuitive yet precise graph-based semantics for the configuration of components and connectors.

The COMMUNITY Workbench is being developed as a proof of concept of the COMMUNITY framework, providing a graphical integrated development environment to write components, draw configurations, and execute the resulting system.

Keywords: Separation of concerns; Computation; Coordination; Distribution; Mobility; High Level Design

1. Introduction

COMMUNITY was initially developed to show how programs fit into Goguen's categorical approach to General Systems Theory, but it is now a platform for research into formal aspects of Software Architecture. It draws ideas from category theory, parallel program design languages, coordination languages, and architecture description languages to obtain a framework for the formal specification of open, reconfigurable, mobile systems in which the computation, coordination, and distribution dimensions are explicitly separated.

The COMMUNITY Workbench is a partial implementation – e.g., it does not include higher-order connectors [1] or reconfiguration [2] – of the formal framework in Java, hiding the categorical underpinnings from the user. It provides an integrated graphical environment to specify components and connectors and configure them into systems which can then be executed.

This paper provides a brief overview of COMMUNITY and the Workbench, using as running example a simple client–server system in which the client is a laptop and the server is a portable printer. The Postscript and PDF files

☆ The development of this tool was supported by Fundação para a Ciência e Tecnologia through the Ph.D. Scholarship SFRH/BD/6241/2001 of Cristóvão Oliveira and through the project POSI/32717/00 (FAST – Formal Approach to Software Architecture), and by the European Commission through the project IST-2001-32747 (AGILE – Architectures for Mobility) of the Global Computing Initiative.

* Corresponding author.
E-mail address: co@di.fct.unl.pt (C. Oliveira).
URLs: http://ctp.di.fct.unl.pt/~co (C. Oliveira), http://mcs.open.ac.uk/mw4687 (M. Wermelinger).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2006.09.005

produced by the user are sent asynchronously via a buffer to the printer. Printing is only possible when the printer and the laptop are in the vicinity of each other, otherwise the files are buffered in the laptop. To simulate the fact that the user carries the laptop and the printer around, the location of the printer is put under the control of the environment and the printer follows the laptop. A more substantial example can be found in [3] where we used COMMUNITY to model the GSM handover protocol that takes place when a mobile phone moves from the scope of one antenna to another one.

The next section provides an overview of COMMUNITY, illustrating the three architectural dimensions with the running example. A detailed and formal account is beyond the scope of this paper; the interested reader is referred to [4,5]. Additionally, the COMMUNITY website [6] provides an updated and general account of the various research strands. In particular, COMMUNITY has been recently radically modified to include event-based interactions [7], but we do not cover those extensions in this paper.

Section 3 then gives a brief account of the Workbench's features, mirroring the structure of Section 2 but also including details on executing the specification of the printing system. A detailed step-by-step guide on how to write, interconnect, and execute the components of the printing system is part of the user manual that comes with the Workbench.

2. CommUnity

COMMUNITY components are in the style of UNITY programs [8], but they also combine elements from IP [9]. This facilitates the description of complex computations at a higher level of abstraction than with the process calculi used by other formal approaches to Software Architecture, like Wright [10] or Darwin [11].

Moreover, COMMUNITY has a richer coordination model than UNITY and IP and, even more important, it requires interaction between components to be made explicit. In this way, the coordination aspects of a system can be separated from the computational aspects and externalized, making explicit the overall configuration of the system in terms of its components and its interactions. A configuration has a precise mathematical definition in terms of an underlying categorical diagram. Such a diagram can be transformed into a single, semantically equivalent, component that represents the whole system. The system can hence be animated by executing that component.

In order to model systems that are location-aware, COMMUNITY adopts an explicit representation of the space within which movement takes place, but does not assume any specific notion of space: the designer has to define it for the application in hand. Data and code mobility is achieved by updating the locations over which code and data are distributed in the defined space.

2.1. Distribution

The space in which the computational elements of the system are distributed is specified by defining the type `Location`. To keep the running example simple we just assume a linear space with integer coordinates, but we could have used for example an array of three real numbers to represent a 3D space.

The relevant properties of the space are captured by two binary relations over locations. Relation *touches* indicates that two positions in the space are "in touch" with each other. Communication among components takes place only when the locations of the data and code involved in the communication are "in touch". Communication in COMMUNITY is achieved via bi-directional sharing of channels and synchronisation of actions (see Section 2.3); hence *touches* must be reflexive and symmetric. For example, we define two positions to be in touch if they are at most one length unit apart. The second relation *reaches* holds when one position is reachable from the other. Movement of data or code to a new position is possible only when that position is reachable from the current one. The relation must be reflexive. For the printing system, we define l_2 to be reachable from l_1 if $l_2 \geq l_1$ (i.e., backward movement is not allowed in the linear space of our example).

The specification of distribution (through type `Location` and relations *touches* and *reaches*) is separate from the specification of computation to allow reuse of components in different topological contexts.

2.2. Computation

The conceptual unit of computation is a COMMUNITY *design*. It consists of a set of typed variables and a set of actions. Variables typed with `Location` are called *location variables*, while the others are called *channels*.

```
design laptop
inloc ll
out outfile@ll : enum(ps, pdf)
prv saved@ll : bool
do edit@ll : true, false -> saved := false
[] save_ps@ll : ~saved -> outfile := ps || saved := true
[] save_pdf@ll : ~saved -> outfile := pdf || saved := true
[] send@ll : saved -> skip

design printer
inloc lp
in infile : enum(ps,pdf)
prv busy@lp : bool; printfile@lp : enum(ps,pdf)
do get@lp : ~busy -> printfile := infile || busy := true
[] prv print@lp : busy -> busy := false
```

Fig. 1. The `laptop` and `printer` components.

COMMUNITY is independent of the actual data types used, but the Workbench provides a fixed set of types: integer and real numbers, booleans, lists, arrays, records, and enumerations. There are *input*, *output* and *private* variables. Input variables are read-only and their value is under the control of the environment. A design with input channels is *open* in the sense that it needs to be connected to other components of the system to read data, as explained in Section 2.3. Output and private variables are called *local* variables and cannot be changed by the environment. Output variables can be accessed by the environment while private ones cannot. Each local channel c is associated with a location variable that holds the position where the value of c is stored.

To illustrate the concrete syntax of COMMUNITY, Fig. 1 shows the two main components of the printing system. Design `laptop` has an input location variable `ll` (location of laptop), an output channel `outfile` which holds a Postscript or PDF file, and a private boolean channel stating whether the file is saved. Both local channels are stored at `ll`. Notice that the locations of the laptop and printer are given by input location variables, as neither of them controls its own position.

There are *private* and *public* actions; only the latter can be coordinated with public actions of other components. Each action has a name and a set of distributed bodies, i.e., each body is associated with a location variable holding the position where the body's code is stored and executed. Each body has two guards and a set of assignments — we write `skip` to denote the absence of assignments. The *safety* and *progress* guards are propositions over channels and location variables and establish an interval in which the enabling condition of the action must lie, the safety guard being the lower bound. The maximal interval is obtained by setting the safety guard to true and the progress guard to false. When the guards are equivalent, we write only one of them.

Mobility is explicitly specified by assignments to location variables: all data and code associated with the location variable on the left-hand side of the assignment are moved to the position given by the right-hand side expression.

At each execution step, one of the actions is non-deterministically chosen and the assignments are executed atomically in parallel if the following conditions hold in the current state:

(1) the location of each body of the chosen action is in touch with all the locations of the other bodies and of the channels occurring in the body's guard and assignments;
(2) all the bodies' enabling conditions hold in the current state;
(3) every assignment to a location variable changes its value to a position that is reachable from its current value.

The last condition directly reflects the mobility constraints imposed by the *reaches* relation, while the first condition reflects the communication and access constraints imposed by the *touches* relation: if a body executing at some location is accessing a channel at another one, then both locations must be in touch, and if an action is distributed, then all its bodies must be in touch with each other for the action to be executed.

In Fig. 1, every action has a single body, and the order of execution is constrained by the use of state channels (`saved` and `busy`). Each design is centralised, in the sense that all its data and code are located in the same place, and all actions are public (i.e., may be coordinated with others) except action `print`. Notice also that `printer` makes a

local copy of the received file because input channels are controlled by the environment and may therefore change their value anytime.

2.3. Coordination

In COMMUNITY, the model of interaction between components is based on synchronisation of public actions and the sharing of input variables of a design with input or output variables of the same type of other designs. Notice that private actions and channels are not involved in interactions. Moreover, it is not allowed to (in)directly share two output variables, or synchronise two actions of the same component. COMMUNITY, unlike some other languages, requires interactions between components to be made explicit through name bindings, because the scope of each name is local to the component where it occurs. Coordination is separate from computation because the name bindings are specified independently of the actions' bodies.

For our example, we could define that `laptop` sends the file to `printer` by binding `outfile` with `infile` and `send` with `get`. This would result in synchronous transmission of the produced files, one-by-one; the synchronised action `send/get` can only execute when the file has been saved, the printer is not busy, and their locations are in touch. The last condition stems from the fact that action `send/get` is distributed over `ll` and `lp`. Moreover, `get` at `lp` must access `infile` which is in fact `outfile` stored at `ll`. To make sure such distributed execution and channel access is always possible, the easiest solution would be to share (i.e., bind) the two location variables to indicate that laptop and printer are always co-located.

If the interaction between components requires additional behaviour, like the buffered asynchronous file transmission for our example, then the proper architectural solution is to use a *connector*. A connector comprises a *glue* and one or more *roles*. The roles are a kind of "formal parameter" of the connector, restricting the components to which the connector can be applied, while the glue is the "body" of the connector, describing how the activities of the roles are to be coordinated. In COMMUNITY, a connector is a star-shaped configuration in which glue and roles are designs and the glue is connected to each role by some name bindings.

For the printing system we need two connectors. The first one is an asynchronous file passing connector with two roles, a sender and a receiver, and a glue providing a buffer co-located with the sender. Fig. 2 shows the involved designs and bindings. The connector consists of one instance of each design, attached such that the output of the sender is the input of the buffer and vice versa for the receiver. Notice that the use of the non-deterministic assignment `oneof`: the role requires the sending component to have an action to produce a file, but does not constrain the production policy. Further note that the receiving component must have an action to synchronise with the buffer's `get` operation. The location of the receiver is left unspecified.

The second connector defines a generic mobility pattern: whenever one component (called the "chaser") is not co-located with the "chased" one, it is moved to the chased component's location. The definition is given in Fig. 3: the roles just impose the components to have a location variable that will be read in the case of the chased component and controlled in the case of the chaser.

A connector is applied by refining each role with a component. The formal definition of *refinement* of designs can be found in [4], but it basically consists in a mapping from the role's names to the component's names and a translation of terms, such that the behaviour is preserved. For our example, roles `sender` and `chased` are refined by `laptop`, and roles `receiver` and `chaser` are refined by `printer`. Just as an illustration, the refinement of `sender` is given by mapping `produce` to `save_ps` and `save_pdf`, `send` to `send`, `outfile` to `outfile`, `ls` to `ll`, and `ready` to `saved` (Fig. 7, right). As can be seen, an action of the role can be refined by multiple actions of the component.

An architectural *configuration* is obtained by selecting components and connectors, creating as many instances of each as needed, and interconnecting component instances with direct name bindings or through the application of connector instances. A configuration has a very precise mathematical semantics, given by a diagram in a category whose objects are the designs and whose morphisms capture a notion of program superposition [8]. Any such categorical diagram can be transformed, by a universal construction called *colimit*, into a single design that represents the whole distributed system [12]. Although the colimit is calculated by our tool, it is important for the user to have an intuitive grasp of how it is computed in order to better understand the trace of its execution.

The variables of the colimit are the union of the variables of all the design instances in the configuration, with shared variables merged into a single one; the result is an output variable only if one of the shared variables is. The actions of the colimit are obtained by taking at most one action from each design instance and synchronising them;

```
connector async {
glue
  design async
    inloc lb
    in infile : enum (ps, pdf)
    out outfile@lb : enum (ps, pdf)
    prv ready@lb : bool; buffer@lb : list(enum(ps, pdf))
    do put@lb : length(buffer) < 3 -> buffer := buffer:<infile>
    [] prv next@lb : buffer /= nil & ~ready -> ready := true
                  || outfile := head(buffer) || buffer := tail(buffer)
    [] get@lb : ready -> ready := false
roles
  design sender
    inloc ls
    out outfile@ls : enum (ps, pdf)
    prv ready@ls : bool
    do produce@ls : true, false -> ready := true
                 || outfile oneof enum(ps,pdf)
    [] send@ls : ready, false -> skip
  design receiver
    in infile : enum (ps, pdf)
    do receive : true, false -> skip
configuration
    s:sender r:receiver a:asynchronous
    attachments {
      a.infile - s.outfile  a.put - s.send    a.lb - s.ls
      a.outfile - r.infile  a.get - r.receive
}   }
```

Fig. 2. The asynchronous file transmission connector.

```
connector follow {
glue
  design chase
    inloc lchased
    outloc lchaser
    do prv move@lchaser : lchaser /= lchased -> lchaser := lchased
roles
  design chaser inloc l
  design chased inloc l
configuration
  c:chase c1:chaser c2:chased
  attachments { c.lchased-c2.l c.lchaser-c2.l }
}
```

Fig. 3. The `follow` connector.

synchronisations imposed by the configuration must be obeyed, but if an action is "free" then it can co-occur with any other actions from other design instances. Private actions can only co-occur with other private actions due to fairness requirements. Synchronising actions amounts to taking the union of their bodies. The colimit, which is unique up to renaming of its variables and actions, avoids clashes between equal names of different design instances.

The variables and actions of the colimit of the printing example are shown in Fig. 10. Unique names are generated by appending the configuration nodes' numbers to the original names. The names of synchronised actions are the

concatenation of the unique names of the original actions. The location variables of all the components, glues and roles have been merged into two: the output variable `lchaser_4` that controls the printer's movement to follow the laptop, and the input variable `lchased_4`, controlled by the user, defining where laptop and buffer are located. As an example of action synchronisation, note how the printing action can occur alone, simultaneously with the printer's movement, together with the `next` action of the buffer, or all three actions can occur synchronously. Actions involving the printer and the laptop, in particular its buffer, are shown to be distributed over both locations.

The size of the colimit is combinatorial in the size of the instances, because it amounts to taking the cartesian product of the actions, removing combinations that are prevented by the explicit synchronisations imposed by the configuration. In other words, if the configuration has n nodes (design instances) with $a_1, \ldots, a_n$ actions, in the limit (i.e., if there are no bindings) the colimit will have $a_1 \times \cdots \times a_n$ actions, and the "size" of each synchronised action (in terms of number of assignments) is the sum of the sizes of the actions that are synchronised.

Given that COMMUNITY is intended for formal research and not for industrial size applications, we have found so far no limitations with our toy examples. Nevertheless, a different approach has also been taken [13], by translating an architectural configuration into a set of mobile agents specified in X-Klaim [14]. This avoids the combinatorial explosion of actions, but requires the encoding of a distributed action execution algorithm that takes into account the conditions mentioned at the end of Section 2.2.

3. Workbench

The Workbench requires a Java 1.4 runtime and is available as a self-installing application from the first author's webpage. The installer was made for multiple platforms (Windows XP, Mac OS X, and Linux) with InstallAnywhere 5.5.1 Now [15]. Additionally, we used JavaCC 2.0 [16] to generate the COMMUNITY parser. The Workbench distribution includes some examples and a user manual.

The current version of the tool allows the user to:

- specify the location type and the relations *touches* and *reaches*;
- write COMMUNITY designs;
- define architectural connectors graphically;
- define the architecture graphically and calculate its colimit;
- animate a design (in particular the colimit of a configuration).

As shown in Fig. 4, the Workbench has three panes. The left pane provides a tree view of the opened configurations (i.e., the architecture and the connectors): there is a leaf for each component of the architecture or design used by a connector, and a leaf for the colimit of the configuration. The right pane has one tabbed window for each item on the left pane that is being worked on. In the case of a design, the window is a simple text editor; in the case of a configuration, the window allows to draw graphs. The bottom pane is the message area. Components, connectors, and architectures can be separately saved and loaded in the Workbench, to allow being reused or adapted.

3.1. Distribution

The definition of the space can be done at any point before animation. There is a dialog (Fig. 5) for the designer to write the location type in the first blank text field in terms of COMMUNITY's pre-defined types (arrays, real numbers, etc.); for the example we used the integer type. In the second and fourth fields, the designer enters partial definitions of the *touches* and *reaches* relations over two implicitly pre-defined location variables (`x` and `y`). In the third field, the Workbench computes the reflexive and symmetric closure of *touches*, and in the fifth field the tool adds the reflexivity condition to *reaches*. Those will be the actual boolean expressions used to evaluate the relations.

3.2. Computation

Designs are written within a simple editable text box and can be checked for syntactic and semantic errors (undefined or duplicate names, type mismatches in expressions and assignments, assignments to input variables, etc.). Designs, being computational units, can be animated as explained in Section 3.4, before being configured into an architecture or connector.

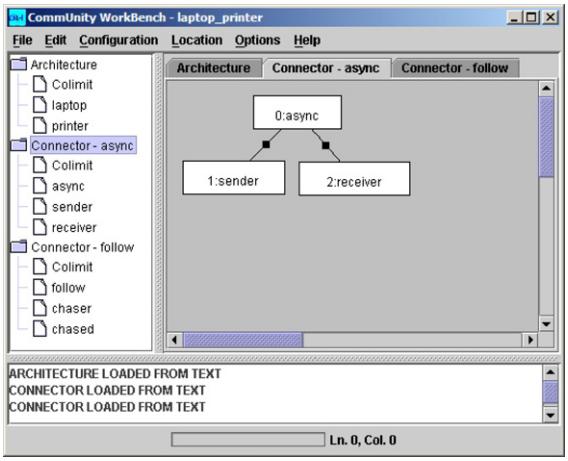

Fig. 4. The async connector.

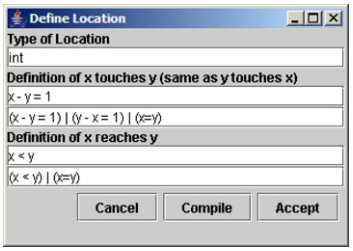

Fig. 5. The location editor.

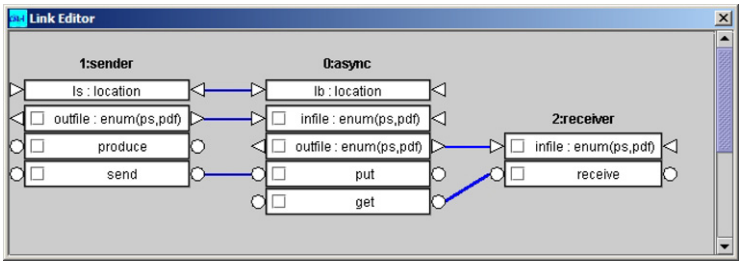

Fig. 6. The link editor.

3.3. Coordination

The two kinds of configurations, connectors and architectures, are both edited in tabbed graphical windows. Nodes in a configuration are uniquely numbered, and they can be dragged around to change the layout, which is saved together with the configuration.

A connector is created by inserting instances of existing designs, and then binding the names of their variables and actions with the graphical link editor (Fig. 6). The editor presents to the user the interface (i.e., the non-private names) of the selected design instances: circles represent actions, inward pointing triangles denote input variables, and outward pointing triangles correspond to output variables. Sharing and synchronisations are defined simply by drawing arcs between variables and actions. Invalid bindings (see Section 2.3) are rejected by the tool. The Workbench also checks that one design is connected to all the other ones, so that the glue can be distinguished from the roles; if there are only two nodes, the user is asked which one is the glue.

An architecture is created in the same way, with the additional possibility of inserting instances of existing connectors and specifying which components refine which roles in what way. The window to add connectors (Fig. 7, left) has one pane for the glue and one for each role. In a role pane the user chooses on the right the component

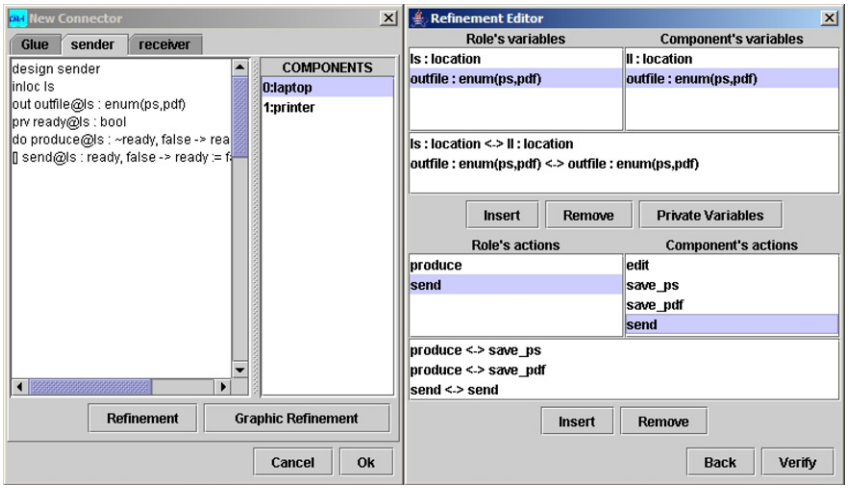

Fig. 7. Refining `sender` with `laptop`.

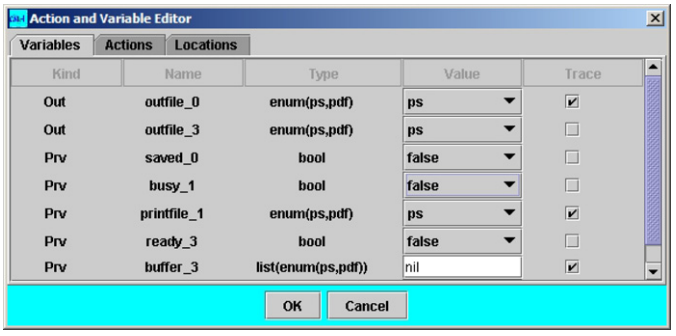

Fig. 8. The action and variable editor.

instance that will refine the current role and clicks the "Refinement" button. A dialog appears (Fig. 7, right) where the user may map each variable/action of the role to the refined variable/action of the component. The workbench prevents wrong refinements, like refining an input variable by an output variable. If the role has no private actions or channels, then a graphic link editor can be used by clicking on the "Graphic Refinement" button.

There is a menu item to compute the colimit of a configuration, be it a single connector or a complete architecture. Being a design, the colimit can be opened in an editing window for inspection.

3.4. Execution

Before running a design, the user has to provide the initial values for all variables, and to choose which variables and actions should be traced (Fig. 8).

The trace window (Fig. 9) gives great flexibility to test several scenarios. Public actions can be selected for execution explicitly by the user or automatically by the tool, in fair or random mode. Private actions are always selected in a fair mode. At each execution step there are three choices for the value of any input variable that is not connected to an output variable: it may be the same as in the previous execution step; it may be a random value; or it may be set by the user.

In Fig. 9 we show part of an execution of the colimit or our example architecture. In the traced scenario, the location `lchaser_4` of the printer is not in touch with the location `lchased_4` of the laptop; hence, the buffer stores the files to be printed (steps 1–8). When the buffer is full, it is only possible to edit and save the file (steps 9–10) because the printer cannot move forward towards the laptop, which is behind the printer. When we change manually the laptop's location to 5, the movement action of the `follow` connector can be executed (step 11) because location 5 is reachable from the printer's location 3. Then the buffer sends to the printer the documents it has (step 12 and following).

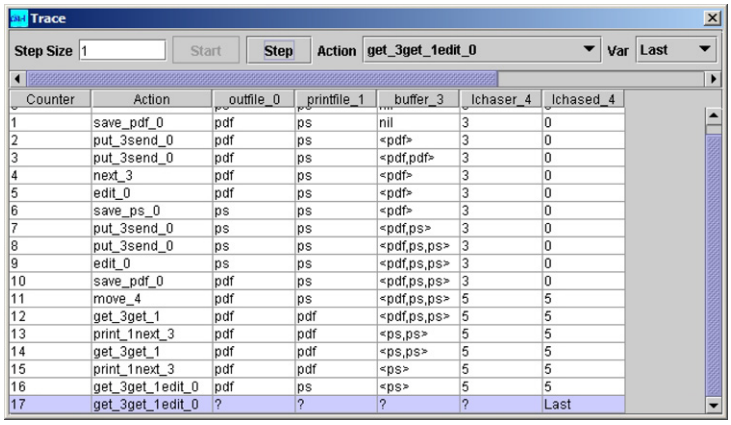

Fig. 9. The trace window.

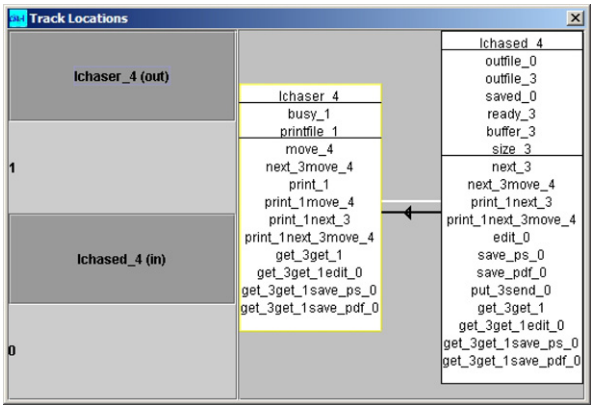

Fig. 10. The location tracker.

During the trace the user can track the location variables in another window (Fig. 10): the current value of each location variable is given on the left, the boxes show the channels and actions stored at each location, the white arc shows that the two positions are in touch, and the black arrow shows the possible direction of movement, i.e., the location of the arrow's target is reachable from the position of the arrow's source. In the example, we see that printer and laptop are in touch, but it is the laptop that can be moved to the printer's location.

4. Concluding remarks

COMMUNITY is a platform for the formal definition of concepts needed for the architectural design of mobile systems. It uses category theory, in particular the notion of colimit, as the mathematical basis for the semantics of configurations, it uses a parallel program design language for the expressive and intuitive definition of computations, it separates coordination from computation as advocated by coordination models, and it encapsulates interactions into connectors as put forward by software architecture. Furthermore, COMMUNITY allows the distribution aspect to be specified independently, but action execution takes into account the topological constraints on vicinity and reachability.

The COMMUNITY Workbench provides a proof of concept for the approach summarised above through an integrated environment that hides the categorical details from the user. The Workbench allows the separate definition of computation, coordination and distribution through distinct windows and dialogs, and the animation of the overall system. The architecture can be checked incrementally at various levels: linguistic errors in designs, invalid bindings and refinements in configurations, and unexpected behaviour during animation. For the latter, it is possible to probe specific scenarios by defining at each step the input values and the action to be executed. The environment also allows the separate saving of designs, connectors and architectures in a textual format that allows multiple users to reuse and adapt them for their needs.

All these features allow users to design architectural models of mobile systems and validate their behaviour. Nevertheless, the transitive nature of bindings and the combinatorial nature of colimits makes it sometimes difficult to spot certain design issues just by tracing the execution. From our experience in using the tool, we found that it would be useful during the design phase to be able to know which actions *cannot* occur simultaneously and which ones *must* occur together, which locations might have to be in touch with each other, and which code and data (i.e., actions and channels) is moved by which actions, among other things. Such checks enable the user to detect potential problems (e.g., a design with too many constituents at the same location), and take corrective actions (e.g., by relocating some of the constituents).

Architectural views are ideally suited to spot such problems, because each view describes a part of the system according to some perspective of interest, making some aspects explicit, while omitting others. We have therefore defined architectural views that highlight the most important aspects for COMMUNITY: computation, coordination and distribution [17]. For example, the coordination view might show a large set of actions that always execute synchronously, whereas the distribution view shows if they are scattered among many locations or not. The workbench's user manual shows what information each view displays. For future work we intend to make it possible to edit the architecture via the views, propagating a change on one view to the other ones.

Acknowledgments

We thank Antónia Lopes and José Fiadeiro for their support and feedback on the tool. We also thank João Feliciano and Helder Neto for implementing the initial version of the workbench, José Aires Camacho for implementing some new features and the output of the textual specification, Mário Costa for the graphical link editor, and Brian Plüss and Frederico Palma for implementing the distribution and mobility aspects.

Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.scico.2006.09.005.

References

[1] A. Lopes, M. Wermelinger, J. Fiadeiro, Higher-order connectors, ACM Transactions on Software Engineering and Methodology 12 (1) (2003) 64–104.
[2] M. Wermelinger, J.L. Fiadeiro, A graph transformation approach to run-time software architecture reconfiguration, Science of Computer Programming 44 (2002) 133–155.
[3] C. Oliveira, M. Wermelinger, J.L. Fiadeiro, A. Lopes, Modelling the GSM handover protocol in CommUnity, in: Proc. of the 2nd Workshop on Formal Foundations of Embedded Systems and Component-Based Software Architectures, Electronic Notes in Theoretical Computer Science 141 (3) (2005) 3–25.
[4] J.L. Fiadeiro, A. Lopes, M. Wermelinger, A mathematical semantics for architectural connectors, in: Generic Programming, in: LNCS, vol. 2793, Springer-Verlag, 2003, pp. 190–234.
[5] A. Lopes, J. Fiadeiro, M. Wermelinger, Architectural primitives for distribution and mobility, in: Proc. 10th Symposium on the Foundations of Software Engineering, ACM Press, 2002, pp. 41–50.
[6] The CommUnity website. URL: http://www.fiadeiro.org/jose/CommUnity.
[7] A. Lopes, J.L. Fiadeiro, A Formal Approach to Event-Based Architectures, in: Proc. Fundamental Approaches to Software Engineering, in: LNCS, vol. 3922, Springer, 2006, pp. 18–32.
[8] K.M. Chandy, J. Misra, Parallel Program Design—A Foundation, Addison-Wesley, 1988.
[9] N. Francez, I. Forman, Interacting Processes, Addison-Wesley, 1996.
[10] R. Allen, D. Garlan, A formal basis for architectural connection, ACM TOSEM 6 (3) (1997) 213–249.
[11] J. Magee, J. Kramer, D. Giannakopoulou, Behaviour analysis of software architectures, in: Software Architecture, Kluwer Academic Publishers, 1999, pp. 35–50.
[12] J.L. Fiadeiro, Categories for Software Engineering, Springer, 2004.
[13] C. Oliveira, A. Lapadula, Distributed execution of CommUnity using Klaim, Unpublished Technical Report, Apr. 2005.
[14] L. Bettini, R. de Nicola, Mobile distributed programming in X-Klaim, in: Formal Methods for Mobile Computing, Advanced Lectures, in: LNCS, vol. 3465, Springer, 2005, pp. 29–68.
[15] Zero G Software, InstallAnywhere. URL: http://www.zerog.com/.
[16] Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator. URL: https://javacc.dev.java.net/.
[17] C. Oliveira, M. Wermelinger, A Model-Driven Approach to Extract Views from an Architecture Description Language, in: Proc. 6th Working IFIP/IEEE Conf. on Software Architecture, IEEE, 2007.